Int. J. Adv. Eng. Pure Sci. 2021, 33(1): 73-82
DOI: 10.7240/jeps.737786

RESEARCH ARTICLE / ARAŞTIRMA MAKALESİ

GraphQL Sorgu Oluşturma Sürecinde Kullanılan Araç ve Yöntemlerin Analizi ve İyileştirilmesi

Analysis and Improvement of Tools and Methods Used in GraphQL Query Building Process

İbrahim Enes AYDOĞDU[1], Ali NİZAM[2]

[1] *Fatih Sultan Mehmet Vakıf Üniversitesi, Bilgisayar Mühendisliği Bölümü, 34445, İstanbul, Türkiye*
[2] *Fatih Sultan Mehmet Vakıf Üniversitesi, Bilgisayar Mühendisliği Bölümü, 34445, İstanbul, Türkiye*

Öz
Günümüzde yaşanan teknolojik gelişmeler, İnternete bağlanan toplam cihaz tür ve sayısında büyük bir artışa yol açmıştır. Sunucu makineler daha fazla istek almaya başlamış hem ağ trafiği hem de sunucu yanıt süresi olumsuz etkilenmiştir. Bu sorunları çözmek için Facebook tarafından 2015 yılında duyurulan GraphQL teknolojisi tek bir istekle birden fazla tablo, koleksiyon veya veri tabanına erişim sağlayarak toplu veri sorgulama ve değiştirmeye imkân vermektedir. Bu sayede cihaz başına düşen istek sayısı ve cihazların belleklerinde tutulacak veri boyutu azalır. Ancak GraphQL yeni bir teknoloji olduğundan henüz kod geliştirme sürecini yöneten ve kolaylaştıran araçlar tam olarak gelişmemiştir. Sunucu kısmında sorguları oluşturmak ve çalıştırmak için önemli miktardaki kodun elle yazılması gerekmektedir. Bu da yazılım geliştiricilere önemli bir iş yükü oluşturmaktadır. Bu çalışmada GraphQL sorgu geliştirme süreci, bu süreci kolaylaştırmak veya otomatikleştirmek için kullanılan araçlar, bu araçların kullandığı yöntemler ve sorgu geliştirme maliyetleri analiz edilmiştir. Bu maliyeti azaltmak için kodları otomatik oluşturan bir yöntem önerilmiş ve bu yöntemi kullanan bir araç geliştirilmiştir. Geliştirilen yöntemin etkinliği diğer yöntemlerle karşılaştırılmış, sayısal olarak incelenmiş ve yazılımcıları önemli miktardaki kodu tekrar yazmaktan kurtararak zamandan tasarruf sağladığı görülmüştür.
Anahtar Kelimeler: GraphQL, otomatik sorgu oluşturma, nesne ilişkisel eşleşme, anlık sorgulama, NoSQL

Abstract
Nowadays, as a result of developing technology, increasing device diversity, and the total number of devices connected to the Internet, servers have started to receive more requests adversely affecting both network traffic and server response time. For eliminating these problems, in 2015, Facebook announced GraphQL technology allowing multiple tables, collections, or databases can be accessed instantly via a single request and a single answer. Therefore, the number of requests per device and the size of the data to be kept in the memory of the devices is reduced significantly. However, it is necessary to write code manually to create and run the GraphQL queries on the server part due to the lack of adequate code management and automation tools. Thus, it creates an additional workload for the developer. In this study, we have analyzed the tools used to automate or facilitate the query development process of GraphQL and compared the cost of query development. A new method and tool for generating GraphQL queries have been developed and its effectiveness has been compared to other methods and evaluated quantitively. The results show that the developers save time by avoiding the burden of writing many lines of code.
Keywords: GraphQL, automatic query generation, object-relational mapping, ad-hoc query, NoSQL

I. GİRİŞ

İnternet üzerinde veri haberleşmesi en önemli yazılım işlevlerinden birisidir. Böylesi dağıtık ortamlarda haberleşme için en yaygın kullanılan teknoloji Rest (Representational state transfer) API'dir [1]. Bu API İnternet adreslerine HTTP protokolüyle çağrı yapmak ve istenilen sonuçları JSON (Java Script Object Notation) biçiminde döndürmek için kullanılır. Ancak günümüzde istemci tarafındaki çağrı sayısının, çağrıda kullanılan veri türlerinin, veri ara yüzlerindeki değişimlerin ve ağ trafiğinin artması Rest API'den farklı teknoloji arayışlarını tetiklemiştir [2], [3].

Rest API kullanılarak bir sitedeki içeriğin özel bir kısmına ulaşmak için yüksek işlem maliyetine sahip iki yöntem kullanılır. Birinci yöntemde tüm içerik tek bir istekle veri tabanından istemci bilgisayara indirilip istenen kısım tüm veriden ayıklanır. Bu yöntem, akıllı telefon ve tablet gibi düşük bellek alanlarına sahip cihazlar için önemli bir veri saklama sorunu oluşturmaktadır. İkinci yöntemde sunucuya birden fazla istek gönderilir. Çok sayıda isteğin eş zamanlı olarak sunucuya gönderilmesi de ağ trafiği ve cevap süresinin artmasına hatta sunucunun aşırı yükten kilitlenmesine yol açabilmektedir.

Sorumlu yazar: Ali NİZAM, Tel: +90 212 521 81 00, e-posta: ali.nizam@fsm.edu.tr
Gönderilme: 16.05.2020, **Düzenleme:** 07.12.2020, **Kabul:** 09.12.2020

Birden fazla alandan veri almak yapısı itibarıyla graf biçiminde modellenebilir [4]. Yukarıda sıralanan sorunları çözmek ve graf yapısının esnekliğinden faydalanmak amacıyla Facebook GraphQL teknolojisini geliştirmiştir [5]. GraphQL, çok sayıda çevrimiçi API ve hizmet tarafından kullanılmaktadır [6], [7]. Drupal kullanım istatistik sitesine göre GraphQL kullanımı 2017 yılından itibaren hızla artmış ve 800 bin civarında projeye ulaşmıştır [8]. REST kullanımı ise 8 milyondan 5 milyon civarına düşmüştür. Ancak akademik açıdan GraphQL hakkında oldukça az araştırma yayınlanmıştır [9].

GraphQL teknolojisi REST gibi bir servis katmanı oluşturmaktan ziyade doğrudan verileri hedefler [10]. Sunucuya gönderilen tek bir istekle ve alınan tek bir yanıtla farklı türlerdeki verilere ulaşmak mümkündür. Bu sayede ağ trafiği ve istemci-sunucu tarafında kullanılan bellek miktarı azalır. Çeşitli araştırmalar özellikle birden fazla kaynaktan veri alınan sorgularda GraphQL'in REST API'ye göre performans avantajı sağladığını göstermiştir [11] [12], [13].

GraphQL kullanarak veri sorgu ve değişiklik işlemlerinin yürütülmesi için şema ve çözümleyici metot tanımlamak gereklidir. Sorgu işlenmesi sürecinde hedef uç noktalardaki (end point) işlemler için gerekli sorgu belgeleri şemaya göre doğrulanır. Şemadaki her nesne bir çözümleyici metotla temsil edilir. Çözümleyici metotlar verinin sorgulanması (query) ve değiştirilmesi (mutation) işlemini gerçekleştirir [9]. GraphQL şemasının oluşturulması ve uygulanmasının test edilmesi çok önemli ve çözüm bekleyen bir sorundur [14]. GraphQL'i veritabanı üzerinde kullanmak isteyen yazılımcılar her koleksiyon için yeniden benzer kodları yazmak zorunda kalmaktadır. Bu yüzden hata oranı artmakta, zaman ve emek israfı oluşmaktadır.

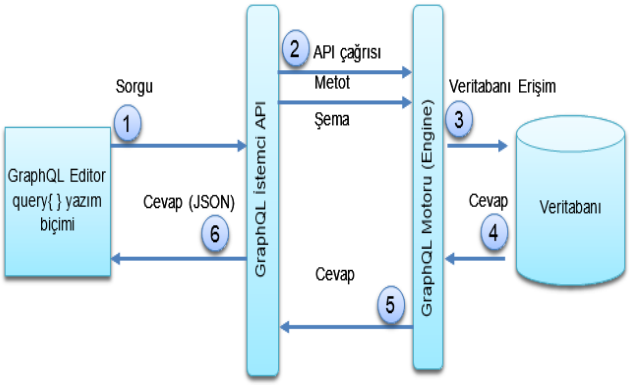
Bu çalışmanın amacı GraphQL sorgularının oluşturulmasını otomatikleştiren, kolaylaştıran, elle kod yazma hatalarını azaltan, emek ve zaman tasarrufu sağlayan araçların kullandıkları yöntemleri mukayeseli bir şekilde incelemektedir. Ayrıca mevcut araçların özellikleri dikkate alınarak GraphQL'in oluşturulmasını kolaylaştıran yeni yöntem önerilmiştir. Bu kapsamda NoSQL (MongoDB) veri tabanları üzerinde GraphQL sorgularının çalışması için gerekli sunucu kodlarını otomatik bir şekilde oluşturan bir altyapı ve bu altyapıyı kullanan bir anlık sorgulama aracı geliştirilmiştir. Bu konudaki literatür özeti Bölüm 2'de sunulmuştur. Bölüm 3'te yeni yöntem ve aracın teknik ayrıntıları açıklanmıştır. Geliştirilen aracın etkinliği MongoDB veri tabanı üzerinde test edilmiş ve bulgular Bölüm 4'te sunulmuştur. Sonuçlar Bölüm 5'te değerlendirilmiştir.

II. MEVCUT YÖNTEM VE TEKNİKLER

Veri tabanı sistemleri ile kodlama dilleri arasında veri yapı ve veri işleme mekanizmaları farklılık gösterir. Bu farkın getirdiği kodlama yükünü azaltmak ve veri akışını yönetmek için ara katman yazılımları kullanılır. Günümüzde yaygın kullanılan ilişkisel veri tabanları için bu çözümler nesne ilişkisel eşleşme (ORM, Object Relational Mapping) ismiyle standart hale gelmiştir [15]. Bu araçlar kullanılarak ilişkisel veri tabanı ile nesne tabanlı diller arasında veri transfer ve dönüşümü için gerekli sınıflar otomatik oluşturulur. Yine ilişkisel veri tabanları üzerinde anlık SQL sorgularını çalıştırmak ve denemek için anlık sorgulama (adhoc query) araçları kullanılır. Ancak GraphQL yeni bir teknoloji olduğu için bu tür araçlar henüz geliştirilme aşamasındadır.

2.1. GraphQL Sorgu Süreci

Bir GraphQL isteği, yazılan sorgunun aşamalı şekilde dönüştürülerek sunucuya gönderilmesi ve cevabın alınması şeklinde yürütülür; bkz. Şekil 1. GraphQL sorguları ön yüzde query{} yazım biçiminde yazılır (1). Query yazım biçimi iç içe sorgu çağrılması ve ortalama-toplam adet gibi grup temelli işlemleri destekleyerek çok önemli bir esneklik sağlar. İstemci API, sorguları arka planda GraphQL sorgu ve değişim işlemleri metotlarıyla eşleştirir ve veritabanı katmanına aktarır (2). Bunu yaparken sorgulanacak verinin yapısını tanımlayan şemaları da kullanabilir. İstemci API sorgu katmanı olmaksızın da doğrudan kullanılabilmektedir. Veri tabanı erişim katmanı gelen istekleri veri tabanı türüne göre SQL veya NoSQL yazım biçimlerinden birisiyle veri tabanına gönderir (3) ve cevabı döndürür (4). Cevap JSON biçimine dönüştürülür (5) ve kullanıcı ara yüzüne aktarılır (6). Bu süreç farklı ürün ve araştırmalarda farklı şekillerde gerçekleştirilebilmektedir.



Şekil 1. GraphQL sorgu süreci

2.2. GraphQL Kod Oluşturma Araçları

Temel kodları Facebook tarafından açık kaynak kodlu sunulmakla birlikte GraphQL hazır bir ürün değil mimari tanımlar içeren ve gerçekleştirme şeklini yazılım firmalarına bırakan bir tarifnamedir (specification). Bu yüzden istemci tarafında ağ isteklerini yönetmek, tampon bellek kullanımı ve kullanıcı ara yüzüne veri aktarımı için birçok araç geliştirilmiştir [16]. Bunların temel yaklaşımları aşağıda sıralanmıştır:

2.2.1. Kodlamayı basitleştiren araç ve yöntemler

Kodlamayı basitleştirme eksenli araçlarda sorgu çalıştırılması için JSON şema ve çözümleyici metotların yazılması yine gereklidir. Ancak bu işlem aracın sağladığı ara depo (repository) ve mesaj sınıfları sayesinde kolaylaştırılır [17]. Bu kapsamda bir ürün olan Apollo, Facebook firmasının React ürünüyle birlikte çalışacak şekilde geliştirilmiştir [18]. Relay ürünü ise React bileşenlerinin ortak bir sorgudan beslenmesini sağlar [19].

2.2.2. Kodları otomatik oluşturmayı sağlayan araç ve yöntemler

Sorgulanmak istenilen veri kaynağının veri yapısını algılayarak otomatik olarak GraphQL sorgu kodları oluşturmayı hedefleyen araçlardır.

- *Kendi API'sini kullanan araçlar*: Bu araçlar standart GraphQL üzerinde kendi özel API yapıları vasıtasıyla sorgu geliştirilmesini sağlar [20]. Önce UML sınıf diyagramları geliştirip, bunları GraphQL şemalarının statik veri organizasyonuyla eşleştirerek kullanmak da önerilmiştir [21]. Bu da önce UML şema oluşturma aşaması gerektirdiğinden maliyetli bir çözümdür. PostgreSQL gibi sadece belirli veri tabanlarına özel şema tanımı ve kodları yazan çözümler de bulunmaktadır [22], [23].
- *Var olan REST API'lerin GraphQL'e dönüştürülmesini sağlayan araçlar*: REST API tanımlarını, GraphQL uç nokta tanımları ve çözümleyici fonksiyonlara dönüştüren araçlardır. Swagger şema şeklinde özel bir girişe ihtiyaç duyabilirler [24].
- *Genel bir haritaya dayalı araçlar*: GraphQL şema ve çözümleyici metot oluşturma işlemini, sorgulanacak kaynaktaki şema yapısı ve GraphQL API arasında oluşturulacak Global Ontoloji Haritaları (GOH) ile sağlamayı öneren araçlar da bulunmaktadır [25]. Bu eşleştirme yöntemiyle GraphQL benzeri teknolojiler için de otomatik kod üretilebilecek ortak bir altyapı oluşturulabileceği ileri sürülmüştür. Ancak henüz bu araçların çalışan bir örneği üretilmediği için geliştirilen yöntemle doğrudan mukayesesi mümkün olmamıştır.

Kodlamayı basitleştiren yöntem ve araçlar GraphQL API'sinin farklı kodlar içerisinden çağrılmasını kolaylaştırır. Ancak sorgu oluşturma aşamasını bütünüyle içermezler. Bu yüzden sadece doğrudan sorgu oluşturmaya yönelik yöntemlerle karşılaştırma yapılmıştır; bkz. Tablo1.

Tablo 1. GraphQL Otomatik Sorgu Oluşturma Yöntemleri Özeti

Yöntem	Dönüşüm Teknolojisi	Platform	Sorgu Biçimi	İlave API Gereği	Veri tabanı Erişimi
Prisma [20]	Veritabanı →Prisma Şema →GraphQL	Genel	Geliştirme Aşamasında (GraphQL Playground)	Şema tanımı	Otomatik oluşturan kütüphane
Hasura.io [23]	Veritabanı →GraphQL	PostgreSQL	Var	Ayrı bir uygulama olarak çalışmaktadır	Otomatik oluşturan kütüphane
StG [24]	Veritabanı→Swagger→GraphQL	Genel	Yok	SwaggerAPI	Var olan erişim kodlarını kullanan bir API oluşturuyor.
PostGraphile [22]	Veritabanı→GraphQL	PostgreSQL	Var	Komut satır ve web için ilave Plug In	Otomatik oluşturan kütüphane
Costal ve diğ.[25]	Veritabanı→GOH→ GraphQL	Genel	Yok	GOH API	GOH

III. ÖNERİLEN YÖNTEM

Bu çalışmada GraphQL sorgu oluşturma sürecini otomatik hale getiren bir yöntem önerilmiş ve bu yöntemi gerçekleştiren bir araç geliştirilmiştir. Yöntem tanımı, sorgulanacak kaynaktaki veri yapısı şemalarının okunmasından sorgu çalıştırılmasına kadar tüm aşamaları içermektedir. Yöntemin geliştirilmesinde kodlamanın otomatikleştirilmesi, platform bağımsızlık, esneklik ve GraphQL API'sindeki değişikliklere kolay uyarlanabilme kriterleri dikkate alınmıştır. Mevcut araçlar incelendiğinde GraphQL API ile sorgu geliştirmede *ara bir dönüşüm teknolojisi* ve *ilave kütüphaneler kullanımı* yöntemi öne çıkmaktadır; bkz. Tablo 1. Ancak bu teknoloji ve kütüphanelerin özel bir yazılım içinde kullanılması, bakım ve yönetimi için ilave emek ve zaman gereklidir. Önerilen yöntemde ise GraphQL için şema ve çözümleyici metot kodları sadece temel GraphQL API'ye bağlıdır. İlave bir sunucu veya kütüphane gerekmeksizin kodlar otomatik oluşturulup sorgu çalıştırılabilir. Böylece GraphQL API'deki olası yenilik ve güncellemelere uyarlamanın kolaylaşması hedeflenmiştir.

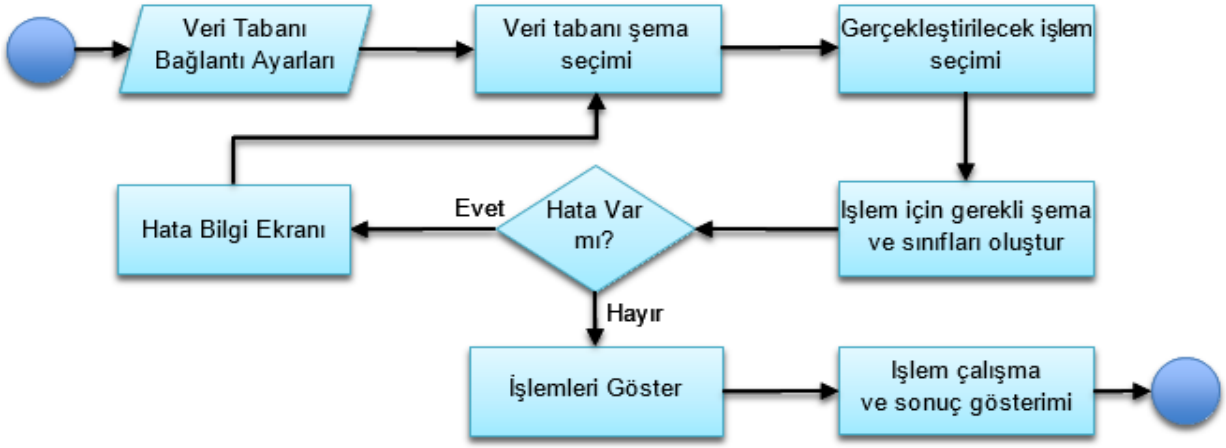
Önerilen yöntemde satır sütun temelinde istenilen kısıtlama ve gruplama işlemlerinin dinamik olarak yapılmasını sağlayan GraphQL Query yazım biçimi desteklenmektedir. Bu sorgu biçimi üzerinden

GraphiQL [26] editörüne dayalı bir anlık sorgu aracı da geliştirilmiştir. Bu sayede aynı istemci metotları kullanılarak çok farklı ve esnek sorgular geliştirilebilir. Anlık sorgu biçimini desteklemeyen araçlarda her işlem için farklı metot yazılması gerekmektedir [20], [24], [25] .

Veri tabanına erişim için gerekli tüm kodun arka planda yazılmasına dayalı nesne ilişkisel eşleşme yöntemleri veri erişimini yöneterek kodlamayı kolaylaştırma gibi önemli faydalar sağlamaktadır. Ancak geliştiriciler arka planda üretilen kodu doğrudan yönetemediği için performans sorunlarına yol açabilmektedir [27]. Bunun yanında *platform bağımlılığı* gibi teknoloji kısıtları da oluşabilmektedir. Örneğin PostGraphile ve Hasura.io gibi ürünlerde sadece üreticinin destek verdiği veri tabanlarına yönelik sorgular hazırlanabilmektedir. Önerilen yöntemde nesne ilişkisel eşleşme yaklaşımı yerine üretilen kodlara açık olarak erişilebilmesi ve gerekirse müdahale edilebilmesi tercih edilmiştir.

3.1. Geliştirilen Aracın Temel Altyapısı

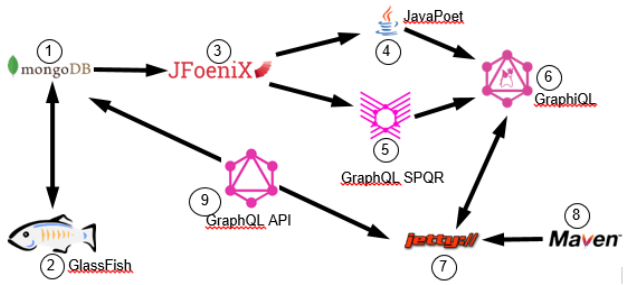
Geliştirilen yöntem, veri tabanındaki şemaların okunmasından kod çalıştırılmasına kadar tüm sorgu oluşturma sürecini otomatik hale getirmektedir; bkz. Şekil 2. Bu yönteme dayalı geliştirilen yazılım aracında ilk aşamada veri tabanlarıyla bağlantı kurulması için gerekli bağlantı ayarları ve veri yapısı seçimleri yapılır. İkinci aşamada kullanıcı seçtiği koleksiyon üzerinde ne tür işlemler yapacağını belirler ve sorgu kodları oluşturulur. Üçüncü aşamada uygulama içerisine eklenen sunucu çalışır ve oluşturulan kodlar bu sunucu üzerinde çalışmaya başlar. Kullanıcı bu kodları kullanarak sorgu, ekleme, silme, güncelleme gibi işlemleri anlık olarak gerçekleştirebilir ve sorgu sonuçlarını görebilir.



Şekil 2. Önerilen yönteme dayalı geliştirilen aracın temel akış diyagramı

3.2. Kullanılan Teknolojiler

GraphQL üzerinde sorgu oluşturmayı otomatikleştirmek ve anlık sorgu aracını geliştirmek için ara yüz, veri tabanı, GraphQL ve otomatik kod oluşturmayı sağlayan teknolojiler entegre edilmiştir; bkz. Şekil 3. Arayüz için JavaFX ve JFoenix, veri tabanı için MongoDB, GraphQL şema ve çözümleyici metotlar için GraphQL API ve otomatik kod oluşturmak için JavaPoet ve GraphQL SPQR teknolojilerinden yararlanılmıştır. Teknolojilerin özellikleri ve kullanım ayrıntıları aynı numaralı alt bölümlerde açıklanmıştır.



Şekil 3. Önerilen yöntemde kullanılan teknolojilerin bütünleşik gösterimi

3.2.1. MongoDB

Geliştirilen sistem farklı NoSQL veri tabanlarıyla çalışabilecek altyapılar içermekle birlikte çalışma kapsamında MongoDB kullanılmıştır. MongoDB, verileri JSON dokümanları halinde saklayan ölçeklenebilir bir NoSQL veri tabanı sistemidir [28]. İlişkisel veri tabanındaki tablolar burada koleksiyona karşılık gelir. İlişkisel tablonun her bir kaydı da bir JSON belgeye karşılık gelir. Tablo sütunu için de belgedeki alanlar kullanılır. Veriler bir belge ağacı şeklinde saklanır. MongoDB'de şema yapısı yoktur ve yazma performansı yüksektir. Şemanın olmaması, her dokümanda farklı alanların tutulabildiği esnek bir yapı sağlar.

3.2.2. GlassFish

GlassFish MongoDB bağlantılarının yönetilmesini sağlar. GlassFish J2EE standardının açık kaynak kodlu temel referans sunucusudur. Düşük kaynak tüketimi ve kolay kurulumu sebebiyle tercih edilmiştir.

3.2.3. JFoenix

Sorgu üretme ve çalıştırma kodları Java dilinde geliştirilmiştir. Geliştirilen aracın ara yüzü için JavaFX altyapısını kullanan ve daha modern bir tasarım sunan

JFoenix tercih edilmiştir. JavaFX içerik, ses, grafik ve video içeren etkileşimli modern uygulamalar oluşturulmasına olanak sağlayan bir Java kütüphanesidir [29].

3.2.4. JavaPoet

JavaPoet, Java kaynak dosyası üretmeyi sağlayan bir kütüphanedir [30]. Bu kütüphane kullanılarak süreç yönetim araçlarının Java ile yazılması gibi çeşitli çözümler geliştirilmiştir [31]. Geliştirilen araçta da GraphQL sunucu kodlarının sınıf, metot ve değişkenlerinin otomatik oluşturulmasında kullanılmıştır; bkz. Şekil 4.

```
ClassName namedBoards = ClassName.get("com.mattel", "Hoverboard", "Boards");

MethodSpec beyond = MethodSpec.methodBuilder("beyond")
    .returns(listOfHoverboards)
    .addStatement("$T result = new $T<>()", listOfHoverboards, arrayList)
    .addStatement("result.add($T.createNimbus(2000))", hoverboard)
    .addStatement("result.add($T.createNimbus(\"2001\"))", hoverboard)
    .addStatement("result.add($T.createNimbus($T.THUNDERBOLT))", hoverboard, namedBoards)
    .addStatement("$T.sort(result)", Collections.class)
    .addStatement("return result.isEmpty() ? $T.emptyList() : result", Collections.class)
    .build();

TypeSpec hello = TypeSpec.classBuilder("HelloWorld")
    .addMethod(beyond)
    .build();

JavaFile.builder("com.example.helloworld", hello)
    .addStaticImport(hoverboard, "createNimbus")
    .addStaticImport(namedBoards, "*")
    .addStaticImport(Collections.class, "*")
    .build();
```

Şekil 4. JavaPoet ile örnek sınıf oluşturma görüntüsü [30]

Şekil 4'teki örnekte bir sınıf nesnesi ve bu sınıf nesnesini oluşturan bir builder nesnesi bulunmaktadır. Oluşturulan sınıf için istenen alanlar belirlendikten sonra bir "builder" nesnesi oluşturulur. Sınıfın kullanacağı kütüphaneler de builder nesnesine "addStaticImport" metoduyla verilir. Böylece sınıf verilecek ek özelliklerle beraber yazıma hazır hale gelir. Örnek kodun çıktısı aşağıdaki gibidir; bkz. Şekil 5.

```
package com.example.helloworld;

import static com.mattel.Hoverboard.Boards.*;
import static com.mattel.Hoverboard.createNimbus;
import static java.util.Collections.*;

import com.mattel.Hoverboard;
import java.util.ArrayList;
import java.util.List;

class HelloWorld {
  List<Hoverboard> beyond() {
    List<Hoverboard> result = new ArrayList<>();
    result.add(createNimbus(2000));
    result.add(createNimbus("2001"));
    result.add(createNimbus(THUNDERBOLT));
    sort(result);
    return result.isEmpty() ? emptyList() : result;
  }
}
```

Şekil 5. Oluşan HelloWorld.java dosyası [30]

3.2.5. GraphQL SPQR

GraphQL SPQR dinamik şema tanımı yapmak için kullanılmıştır. GraphQLSPQR, Java kodundan dinamik olarak GraphQL şeması oluşturmak için kullanılan bir Java kütüphanesidir. Sunucu kodlarına yerleştirilen notlarla (@Annotation) bir şema oluşturur. Kullanıcının şemayı elle yazmasına gerek kalmaz. Aşağıdaki örnekte istemciye bütün bağlantıları çekme izni "@GraphQLQuery" notu ile verilmiştir; bkz. Şekil 6. Bu not herhangi bir metoda verildiğinde GraphQL artık onu bir sorgu metodu olarak görür. Insert, Update, Delete gibi işlemlerin metotları ise "@GraphQLMutation" notuyla tanıtılır. Metot parametreleri de "@GraphQLArgument" ile verilir. Argüman adı, verilecek parametre adıyla aynı olmalıdır.

```
public class Query { //1

    private final LinkRepository linkRepository;

    public Query(LinkRepository linkRepository) {
        this.linkRepository = linkRepository;
    }

    @GraphQLQuery //2
    public List<Link> allLinks(LinkFilter filter,
                               @GraphQLArgument(name = "skip", defaultValue = "0") Number skip, //3
                               @GraphQLArgument(name = "first", defaultValue = "0") Number first) {
        return linkRepository.getAllLinks(filter, skip.intValue(), first.intValue());
    }
```

Şekil 6. Örnek GraphQL SPQR görüntüsü [32]

3.2.6. GraphiQL

Facebook tarafından GraphQL için istemci tarafında veri sorgulama, ekleme, silme vb. gibi işlemleri yapabilmek için tasarlanmış resmi editördür [26]. Herhangi bir kuruluma ihtiyaç duymaz. GraphiQL basit bir HTML dosyasından oluşur. Bu dosya eklenecek projeye göre düzenlenerek her türlü projeye kolay bir şekilde eklenebilir. Herhangi bir web tarayıcısı üzerinde çalışabilir. JSON tabanlı sorgu biçiminin kullanılması aynı metodu kullanarak verinin farklı görünümlerini temel API'yi değiştirmeden sorgulama esnekliği sağlar; bkz. Şekil 7.

Şekil 7. GraphiQL'den örnek görüntü [26]

3.2.7. Jetty server

Geliştirilen yöntemde GraphQL sorgu kodlarını çalıştırmak için kullanılmıştır. Jetty, Java Servlet sınıflarını çalıştırmak için geliştirilmiş bir web sunucusudur. En önemli rakibi Apache Tomcat'e göre daha basit bir yapısı vardır ve karmaşık işlemleri yürütmekte daha etkindir [33]. Ayrıca sunucu kütüphanesinin küçük boyutlu olması projelerin içine rahatça eklenebilmesini sağlar. Küçük ve orta ölçekli GraphQL uygulamaları için Jetty kullanımı tavsiye edilmektedir.

3.2.8. Maven

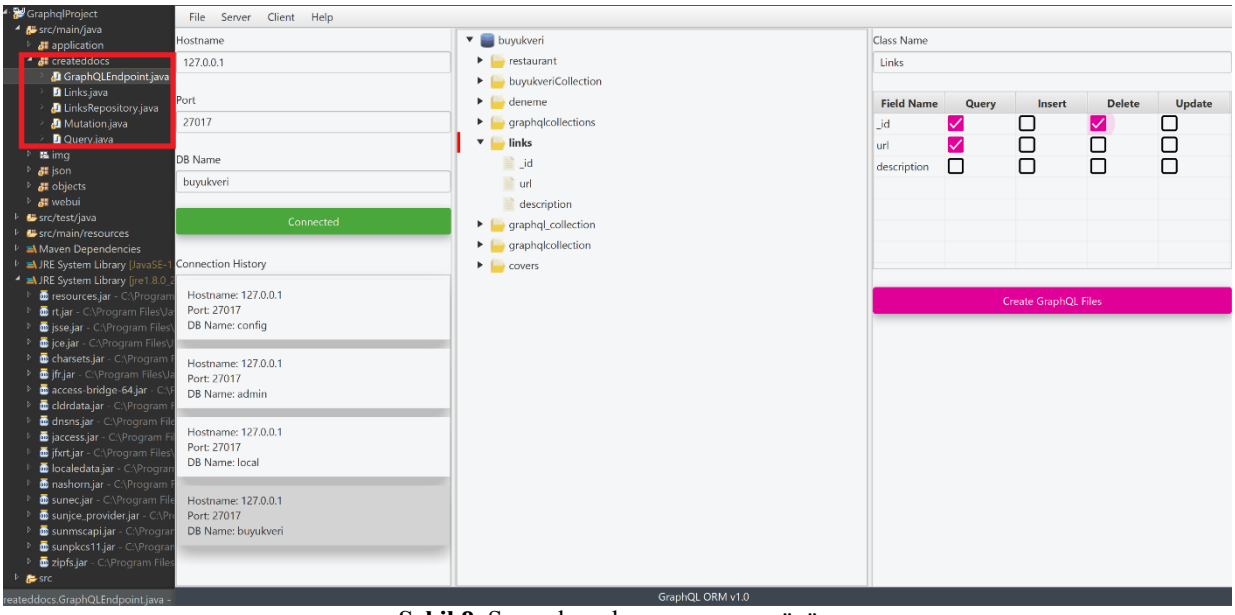
Maven eklentilerinin (plugin) uygulama içerisinden çalıştırılmasını sağlar. Bu kütüphane dinamik olarak oluşturulan GraphQL kodlarının çalıştırılacağı Jetty sunucusunu ayağa kaldırmak için kullanılmıştır.

3.2.9. GraphQL API

GraphQL'in sunucu üzerinde çalıştırılmasını gerçekleştiren yazılım kütüphanesidir. Bir şema ve o şemayı tanımlayan dosyalardan oluşur. İstemci tarafından gönderilen istekleri yöneterek, istemciye en kısa sürede ve en az ağ trafiğiyle sonuçları iletmeyi amaçlar. Birçok programlama dili için kütüphane ve sunucu desteği sağlar. Gelen istekleri sunucu üzerinde karşıladığından tüm veri tabanlarıyla sorunsuz çalışır.

3.3. Sorgu Hazırlama Aracı

Geliştirilen aracın temel işlevleri dört ana kısımda gösterilmektedir; bkz. Şekil 8. Soldan itibaren ikinci çerçevede bağlantı kurulacak veri tabanı için bağlantı ayarları yapılır. "Hostname", "Port" ve "DB Name" değerlerini aldıktan sonra veri tabanına bağlanılır. Bağlantı geçmişi "Connection History" kısmında gösterilir. Buradaki bağlantı ayarları kullanılarak kolay ve hızlı bir şekilde veri tabanına tekrar bağlanmak mümkündür. Üçüncü çerçeve bağlanılan veri tabanındaki koleksiyonları ve içerdikleri alanları bir ağaç yapısında sunar. Kullanıcı işlem yapmak istediği koleksiyonu seçtikten sonra bunlar üzerinde yapılacak işlemleri belirlemek için dördüncü çerçeveye geçer/ilerler. Dördüncü çerçevede, seçilen koleksiyon ve koleksiyonda var olan alanlar listelenerek yapılacak işlemler (sorgu, ekleme, silme gibi) belirlenir. İşlemler belirlendikten sonra "Create GraphQL Files" butonuna tıklandığında gerekli olan sınıflar ve metotlar otomatik olarak oluşturulur, ilk çerçeve açılır ve oluşturulan kod dosyaları burada listelenir. Dosyalar listelendikten sonra araç içerisinde bulunan sunucu otomatik olarak çalıştırılır ve oluşturulan kod dosyaları okunarak GraphQL'in sunucu kısmı çalıştırılır. Oluşturulan kodlar herhangi bir GraphQL sunucusunda çalışabilecek şekilde biçimlendirilmiştir.



Şekil 8. Sorgu hazırlama aracı ara yüzü

3.4. Sorgu Çalıştırma Aracı

Kullanıcı, GraphiQL'e dayalı *sorgu çalıştırma sayfasında* seçtiği özelliklere göre istediği sorguları GraphQL sorgu biçiminde çalıştırıp dönen verileri görebilir; bkz. Şekil 9. Sorgu, arka planda çalışan GraphQL sunucusu tarafından ilgili veri tabanına gönderilir ve sonuçlar ekrana aktarılır. Açılan editörde sorgu kodları üzerinde istenen güncellemeler yapılabilir. Bu yazılıma önemli bir esneklik kazandırmaktadır.

Şekil 9. Sorgu çalıştırma aracı görüntüsü

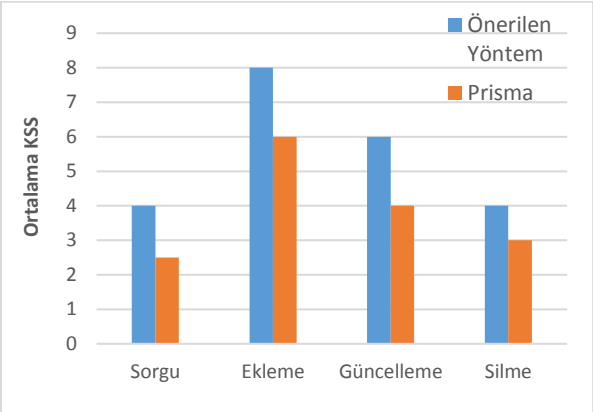
IV. BULGULAR

Bu bölümde GraphQL sorgu oluşturma sürecinde kullanılan farklı sistemler ile önerilen yöntemin etkinliği ve fonksiyonları; ürettikleri kod satır sayıları (KSS), esneklik ve destekledikleri platformlar açısından değerlendirilmiştir. İncelenen yönteme ait niteliklerin birbirini etkileyebileceği göz önüne alınmıştır. Örneğin kodları oluşturmadan doğrudan hazır kütüphane kullanımı esnekliği azaltabilmektedir. KSS somut bir ölçü birimi olması sebebiyle hala yazılım geliştirme sürecinde harcanacak emek ve zamanı ölçmek için en geçerli ölçüdür [34]. Literatürde incelenen GraphQL kod oluşturma sürecini otomatikleştiren bazı araçların kod üretme mekanizmaları erişilebilir değildir. Java tabanlı bir otomatik kod üretme ürünü de tespit edilememiştir. Bu kapsamda kod üretkenlik düzeyini karşılaştırmak için JavaScript kodu üreten Prisma ürününden faydalanılmıştır. Hasura.io ve PostGraphile ürünleri de arka planda benzer kodlar üretmektedir. Ancak bunlar sadece PostgreSQL veri tabanını desteklediği için tercih edilmemiştir. Swagger-to-GraphQL de özel dönüşüm şemaları gerektirdiği için geliştirilen yöntemle doğrudan karşılaştırılması mümkün olmamaktadır. İncelemede MongoDB üzerinde basit veri türüne sahip üç alan içeren bir koleksiyon kullanılmıştır. Bu koleksiyon üzerinde sorgu ve değişim işlemleri yapmak için gereken çözümleyici metotların KSS'leri Tablo 2'de verilmiştir. Kodlar ayrıntılı inceleme yapmak isteyenler için Ek 1'de sunulmuştur.

Tablo 2. Bir koleksiyon üzerinde yürütülen işlemler için gerekli çözümleyici metot kod satır sayısı

	Metot Adeti	Önerilen Yöntem Toplam KSS	Prisma Toplam KSS
Sorgu	4	16	10
Ekleme	1	8	6
Güncelleme	3	18	12
Silme	3	12	9
Ortalama KSS	4,9		3,4

Her bir işlem için ortalama KSS, işlemin toplam KSS'nın metot sayısına bölünmesiyle hesaplanır. Sorgu işlemini gerçekleştirmek için gerekli ortalama KSS açısından Prisma (3,4) ürününün önerilen yönteme göre (4,9) avantajlı olduğu söylenebilir; bkz. Şekil 10. Önerilen yöntemde ayrı ayrı her işlem için de KSS'nın fazla olduğu açıkça görülebilir. Ancak Prisma ürünü çözümleyici metotları otomatik yazmamaktadır. Ayrıca veri tabanı işlemleri için merkezi bir kütüphane kullanmakta ve ayrı ayrı kod oluşturmamaktadır. KSS açısından avantajlı gibi görülen bu durum kod esnekliğini azaltmakta ve kodun farklı platformlara taşınmasını güçleştirmektedir. Benzer bir çözüm olan Hasura'da [23] ayrı bir ürün olarak kurulmakta ve geliştirilen özel yazımlara entegrasyon için ilave işlemler gerektirmektedir.



Şekil 10. Önerilen yöntem ve Prisma ürünü için ortalama KSS

GraphQL sorgularını gerçekleştirmek için yazılması gereken toplam KSS da yazılımcıların iş yükünü belirlediğinden önemli bir inceleme konusudur. Bu kapsamda önerilen yöntemde oluşturulan tanım kodları dahil tüm kodlar toplanmalıdır. Sorgu sınıfı için 34 (Ek 2), değişim sınıfı için 58 (Ek 3), veri şeması tanımı için 29 (Ek 4), GraphQL sunucu ayarları için 29 (Ek 5) ve veri tabanı sunucu tarafındaki işlemler için 93 (Ek 6) satır kod gerekmektedir. İncelenen basit koleksiyon için gerekli sorgu toplam KSS (genel sunucu ayarları hariç) 214'tür. Birçok araştırmanın ortalamasına göre bir programcının aylık üretebileceği kod miktarı ortalama 162-480 (ortalama 366) satır aralığındadır [35]. Sorgu oluşturma kodları birbirine benzediğinden üretkenlik artacaktır. Ancak hata düzeltme ve test gibi işlemler de dikkate alınırsa aylık GraphQL sorgu üretkenliği için üst sınıra yakın bir ortalama olan 500 KSS kabul edilebilir. Bunun neticesinde basit bir koleksiyon için gerekli GraphQL kodlarını otomatik yazmanın yaklaşık 8.5 (20 gün x (214/500) günlük bir emek-zaman tasarrufu sağlayacağı söylenebilir. Karmaşık koleksiyonlarda tasarruf miktarı artacaktır.

V. SONUÇLAR VE ÖNERİLER

Çalışmada GraphQL dili için sorgu oluşturma yöntem ve araçları karşılaştırılarak yeni bir yöntem önerilmiştir. Bu yöntemi gerçekleştirmek için bir otomatik kod üretme ve anlık sorgu aracı geliştirilmiştir. Sorgu oluşturma sürecine ait sayısal sonuçlar bulgular bölümünde sunulmuştur. Buna göre GraphQL sorgu üretme sürecinin yazılımcılar için önemli bir iş yükü oluşturduğu görülebilir. Geliştirilen yöntemi kullanan araç bu işlemi saniyeler içerisinde otomatik yapmaktadır. Böylece maliyet ve zamandan önemli tasarruf elde edilebilir.

Geliştirme sürecinde araçların ürettikleri KSS genel olarak birbirine yakındır. Ancak ORM temelli araçların platform bağımlı oldukları görülmektir. Örneğin PostGraphile ve Hasura ürünleri sadece PostgreSQL veri tabanını destekler. Ayrıca GraphQL temel standardından uzaklaştıkça temel fonksiyonları tekrar sağlamak için ilave araç geliştirilme ihtiyacı da doğmaktadır. Örneğin Prisma ürünü kendine has bir şema tanımı kullanmaktadır. Bu yüzden standart GraphiQL ara yüzü üzerinden esnek sorgu yapılamamaktadır. Firmanın kendi ara yüzü henüz geliştirme aşamasındadır.

Bu çalışmada ilk hedef olarak GraphQL sorgu çalıştırma sürecine ve bu sürecin otomatik hale getirilmesine odaklanılmıştır. Temel sorguların denenmesi için basit bir koleksiyon yapısı kullanılmıştır. İleride yapılacak çalışmalarda bileşik sorgular içeren daha karmaşık koleksiyon yapılarının kod üretme sürecine etkisi incelenebilir. Ayrıca sorguların daha hızlı ve verimli çalışması için gerekli optimizasyon araçları da önemli bir araştırma konusudur. Bu kapsamda farklı yöntemlerin performansları mukayese edilebilir ve performans artışına yönelik yeni yöntemler önerilebilir. GraphQL'in doğrudan graf tabanlı veri tabanları üzerinde çalıştırılması için gerekli sorguların hazırlanma süreci de araştırılacak diğer bir konudur.

TEŞEKKÜR

Bu çalışmanın yapılması için bizleri teşvik eden Fatih Sultan Mehmet Vakıf Üniversitesi Bilgisayar Mühendisliği Bölüm Başkanı Sayın Prof. Dr. Ali Yılmaz Çamurcu'ya teşekkür ederiz.

KAYNAKLAR

[1] Li, L., Chou, W., Zhou, W., & Luo, M. (2016). Design Patterns and Extensibility of REST API for Networking Applications. IEEE Transactions on Network and Service Management, 13(1), 154–167. https://doi.org/10.1109/TNSM.2016.2516946

[2] Ghebremicael, E. S. (2017). Transformation of REST API to GraphQL for OpenTOSCA. https://doi.org/10.18419/opus-9352

[3] Howtographql. (2020). GraphQL is the better REST. Retrieved March 3, 2020, from https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[4] He, H. (2008). Graphs-at-a-time : Query Language and Access Methods for Graph Databases, 405–417.

[5] Facebook. (2015). GraphQL. Retrieved February 28, 2020, from http://spec.graphql.org/July2015/

[6] Hartig, O., & Pérez, J. (2018). Semantics and Complexity of GraphQL Preprint Version *. 27th World Wide Web Conference on World Wide Web (WWW).

[7] GraphQL. (2020). Who's Using | GraphQL. Retrieved March 3, 2020, from https://graphql.org/users/

[8] Drupal. (2020). Usage statistics for GraphQL | Drupal.org. Retrieved March 3, 2020, from https://www.drupal.org/project/usage/graphql

[9] Vogel, M., Weber, S., & Zirpins, C. (2018). Experiences on Migrating RESTful Web Services to GraphQL, 2, 283–295.

[10] Wittern, E., Cha, A., & Laredo, J. A. (2017). Generating GraphQL-Wrappers for REST(-like) APIs. In ICWE 2018. Springer, Cham.

[11] Rasool, S., Khan, R., & Mian, A. N. (2019). GraphQL and DC-WSN-Based Cloud of Things. IT Professional, 21(1), 59–66. https://doi.org/10.1109/MITP.2018.2876982

[12] Taskula, T. (2019). Advanced Data Fetching with GraphQL: Case Bakery Service.

[13] Guo, Y., Deng, F., & Yang, X. (2018). Design and Implementation of Real-Time Management System Architecture based on GraphQL Design and Implementation of Real-Time Management System Architecture based on GraphQL. In IOP Conf. Ser.: Mater. Sci. Eng. (p. 466). https://doi.org/10.1088/1757-899X/466/1/012015

[14] Vargas, D. M., Mayor, U., Sim, D. S., Blanco, A. F., Pablo, J., Alcocer, S., … Bergel, A. (2018). Deviation Testing: A Test Case Generation Technique for GraphQL APIs, 1–9.

[15] Torres, A., Galante, R., Pimenta, M. S., Jonatan, A., & Martins, B. (2017). Twenty years of object-relational mapping : A survey on patterns, solutions, and their implications on application design, 82, 1–18. https://doi.org/10.1016/j.infsof.2016.09.009

[16] Porcello, E., & Banks, A. (2018). Learning GraphQL: Declarative Data Fetching for Modern Web Apps. O'Reilly Media.

[17] Wernet, C. (2017). Unifying access to data from heterogeneous sources through a RESTful API using an e icient and dynamic SQL-query builder. Hochschule Karlsruher Technik und Wirtschaft.

[18] Apollo. (2020). Executing a query. Retrieved February 29, 2020, from https://www.apollographql.com/docs/react/data/queries/

[19] Relay. (2020). QueryRenderer. Retrieved March 2, 2020, from https://relay.dev/docs/en/query-renderer

[20] Prisma. (2020). GraphQL Usage - Prisma. Retrieved March 3, 2020, from https://www.prisma.io/with-graphql

[21] Rodriguez-Echeverria, R., Cánovas Izquierdo, J. L., & Cabot, J. (2017). Towards a UML and IFML Mapping to GraphQL. In ICWE 2017 (pp. 149–155). Springer Verlag. https://doi.org/10.1007/978-3-319-74433-9_13

[22] PostGraphile. (2020). CRUD Mutations. Retrieved March 3, 2020, from https://www.graphile.org/postgraphile/crud-mutations/

[23] Hasura.io. (2020). Realtime GraphQL on PostgreSQL. Retrieved May 10, 2020, from https://hasura.io/

[24] StG. (2020). Swagger-to-GraphQL. Retrieved February 29, 2020, from https://www.npmjs.com/package/swagger-to-graphql

[25] Costal, D., Farré, C., Gómez, C., Jovanovic, P., Romero, O., & Varga, J. (2017). Semi-automatic Generation of Data-Intensive APIs. Retrieved from http://opendata-ajuntament.barcelona.cat/data/en/dataset

[26] Electronjs. (2020). GraphiQL | Apps | Electron. Retrieved March 4, 2020, from https://www.electronjs.org/apps/graphiql

[27] Chen, T. H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. Proceedings - International Conference on Software Engineering, (CONFCODENUMBER), 1001–1012. https://doi.org/10.1145/2568225.2568259

[28] MongoDB. (2020). The database for modern applications. Retrieved March 4, 2020, from https://www.mongodb.com/

[29] JFoenix. (2020). JavaFX Material Design Library. Retrieved May 11, 2020, from http://www.jfoenix.com/

[30] Javapoet. (2020). Javapoet: A Java API for generating .java source files. Retrieved March 4, 2020, from https://github.com/square/javapoet

[31] Kozma, D., Varga, P., & Larrinaga, F. (2019). Data-driven Workflow Management by utilising BPMN and CPN in IIoT Systems with the Arrowhead Framework. IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2019-Septe, 385–392. https://doi.org/10.1109/ETFA.2019.8869501

[32] Howtographql. (2019). Alternative

approaches to schema development. Retrieved March 4, 2020, from https://www.howtographql.com/graphql-java/11-alternative-approaches/

[33] Biying, L. (2010). Jetty improves the performance of network management system based on TR069 protocol. 2010 IEEE International Conference on Intelligent Computing and Intelligent Systems, 3, 799–801. https://doi.org/10.1109/ICICISYS.2010.5658303

[34] McConnell, S. (2006). Software Estimation : Demystifying the Black Art. Microsoft Press, pp 136.

[35] Capers, J., & Bonsignour, O. (2011). The Economics of Software Quality. Addison-Wesley.